Performance Analysis and Optimization of In-situ Integration of Simulation with Data Analysis: Zipping Applications Up*

Yuankun Fu, Feng Li
Purdue University, Indianapolis
Indianapolis, Indiana
{fu121,li2251}@purdue.edu

Fengguang Song
Indiana University-Purdue University
Indianapolis, Indiana
fgsong@cs.iupui.edu

Zizhong Chen
University of California, Riverside
Riverside, California
chen@cs.ucr.edu

ABSTRACT

This paper targets an important class of applications that requires combining HPC simulations with data analysis for online or real-time scientific discovery. We use the state-of-the-art parallel-IO and data-staging libraries to build simulation-time data analysis workflows, and conduct performance analysis with real-world applications of computational fluid dynamics (CFD) simulations and molecular dynamics (MD) simulations. Driven by in-depth performance inefficiency analysis, we design an end-to-end application-level approach to eliminating the interlocks and synchronizations existent in the present methods. Our new approach employs both task parallelism and pipeline parallelism to reduce synchronizations effectively. In addition, we design a fully asynchronous, fine-grain, and pipelining runtime system, which is named Zipper. Zipper is a multi-threaded distributed runtime system and executes in a layer below the simulation and analysis applications. To further reduce the simulation application's stall time and enhance the data transfer performance, we design a concurrent data transfer optimization that uses both HPC network and parallel file system for improved bandwidth. The scalability of the Zipper system has been verified by a performance model and various empirical large scale experiments. The experimental results on an Intel multicore cluster as well as a Knight Landing HPC system demonstrate that the Zipper based approach can outperform the fastest state-of-the-art I/O transport library by up to 220% using 13,056 processor cores.

KEYWORDS

High performance computing, performance analysis and optimization, in-situ/in-transit workflows

1 INTRODUCTION

As high end supercomputing systems are evolving from petascale to exascale, data generated from extreme-scale modeling and simulation applications reach a scale of hundreds of terabytes or even petabytes. The grand challenges confronted by today's scientific computing community hence include not only computation-intensive simulations, but also data-intensive analyses that need to process the huge amount of computed results generated from the simulations [7, 31, 44]. Today, it is even challenging to answer certain basic questions such as: Did any unusual phenomena happen or not during the simulation? When and where did they occur? With the new advanced big data analytics techniques, it is more and more popular and appealing to combine modeling/simulation with big data analysis to create a virtuous cycle that amplifies their collective effects [16, 35, 37].

However, it is exceedingly challenging to achieve high performance for an integrated workflow with both simulation and data analysis applications particularly at extreme scales. There exist workflow solutions that target high productivity. Workflow middleware such as Kepler [28] and Pegasus [10] has been widely used in different scientific domains. They provide orchestrating, executing, and monitoring *coarse-grain steps* in a workflow. Each step runs an application program or web service [17]. Also, those participant steps are often *loosely coupled* such that the resultant workflows have higher latencies (i.e., milliseconds or much more) than the MPI-based HPC applications (i.e., microseconds).

In this paper, we seek to achieve the microsecond-level HPC performance on scientific workflows. Achieving high performance workflows requires we solve the following issues. *First*, what could be the minimum end-to-end time-to-solution and how can we achieve it? *Second*, simulation and data analysis applications work as an interactive producer-consumer system, then how can we reduce the simulation stall time if the analysis is slow? *Third*, how can we reduce the I/O time between simulation and analysis applications? The third issue of I/O bottleneck has been well recognized and studied by many researchers. For instance, in-situ/in-transit approaches, data-staging approaches, and a number of high-level I/O libraries have been developed to reduce the I/O bottleneck. Section 2 will briefly introduce a few state-of-the-art I/O transport libraries.

Instead of focusing on the I/O bottleneck only, we bring on an end-to-end approach to optimizing a scientific workflow's time-to-solution, which is comprised of simulation time, data analysis time, and I/O time. We use the latest high-level I/O libraries such as MPI-IO [45], ADIOS [27], DataSpaces [11], DIMES [50], Decaf [13], and Flexpath [9] to "glue" *standalone* simulation and analysis applications in a workflow. We have developed seven different workflow implementations to combine a lattice Boltzmann method [20] based computational fluid dynamics (CFD) simulation with a turbulence flow analysis application. Each workflow implementation employs a different I/O transport method (in total, seven methods). From the

*This material is based upon research supported by the Purdue Research Foundation and by the NSF Grant# 1522554. ⋄ Corresponding author: Fengguang Song.

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208049

Fu, Y., Li, F., Song, F., & Chen, Z. (2018). Performance Analysis and Optimization of In-situ Integration of Simulation with Data Analysis: Zipping Applications Up. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (pp. 192–205). New York, NY, USA: ACM. https://doi.org/10.1145/3208040.3208049

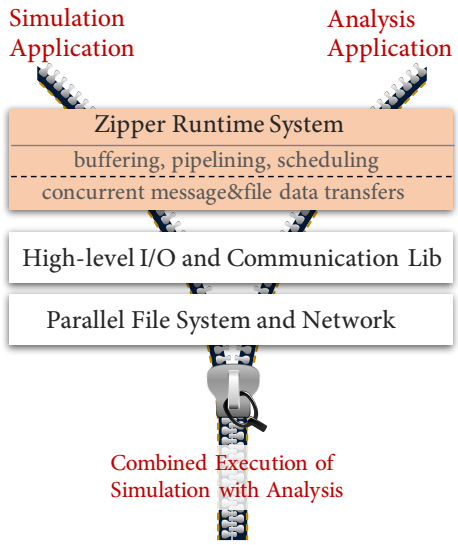


Figure 1: The Zipper runtime system.

experimental results, we find that these workflows' end-to-end time is significantly larger than the essential simulation time or analysis time (shown in Section 3). Detailed performance analysis then identifies a set of performance inefficiencies such as synchronization with centralized servers, coarse-grain critical sections, interlock between applications, barriers, network bandwidth contention, and application stalls.

In order to solve the performance inefficiencies, we devise a new approach, which uses fine-grain data blocks, task parallelism, and pipelining parallelism to tightly interleave simulation and analysis applications. The new approach is driven by data availability, and has no artifactual data dependency (e.g., barriers) between tasks. A runtime system called *Zipper* is designed and developed to enable the new end-to-end approach, as shown in Figure 1. Zipper is located below the application layer, and above the high-level I/O and communication libraries. The Zipper runtime system itself has two strata: 1) The upper stratum provides the functions of buffering data in memory, pipelining data blocks from simulation to analysis applications, and scheduling data transfer operations and data analysis tasks; 2) The lower stratum is an optimization layer, which can transport computed results by two concurrent channels: low-latency HPC network and file-based parallel file system. Section 4 will introduce the Zipper runtime system. In addition to the parallel framework implementation, we also build an analysis model to evaluate the Zipper system. With this performance model, we are able to estimate a combined workflows's time-to-solution, and provide an insight into which component should be improved to achieve the fastest end-to-end time.

We conduct experiments with synthetic applications, a computational fluid dynamics (CFD) application, and a LAMMPS application on two supercomputing systems. The CFD application is coupled with an online statistical turbulence analysis, and the LAMMPS application is coupled with the Mean-Squared Displacement (MSD) data analysis. The paper shows three types of experiment: The first type of experiment is used to validate the analytical performance model; the second type is used to show that the concurrent dual-channel data transfer optimization can reduce data transfer time as well as the simulation application stall time; and the third type validates the scalability of the Zipper system. Based on the experimental results, using Zipper can outperform the fastest state-of-the-art I/O transport library by up to 2.2 times on 13,056 cores. The performance benefits have been studied and analyzed by collecting and comparing different workflow implementations' traces.

To the best of our knowledge, this work makes the following contributions:

- Detailed performance analysis and comparison between an arrange of state-of-the-art I/O transport libraries designed for implementing high performance scientific workflows.
- An end-to-end approach to combining the pipelining parallelism and the asynchronous task parallelism, at a fine-grain task level, to create the new Zipper runtime system for minimized workflow end-to-end time.
- Introducing the concurrent data transfer optimization to reduce I/O time and simulation stall time with in-depth performance analysis.
- Application of the Zipper runtime system to large-scale CFD and molecular dynamics. The experimental results demonstrate Zipper provides better performance than the existing work. Zipper's end-to-end time has also been verified by a performance model and detailed traces.

In the remainder of the paper, the following section introduces different state-of-the-art I/O transport libraries. Section 3 shows performance analysis of scientific workflows using the I/O transport libraries. Section 4 introduces the Zipper runtime system, its parallel implementation, and an analytical performance model. Section 5 compares the existing work with our work. Finally, Sections 6 and 7 present the experimental results and summarize the paper.

2 BACKGROUND OF EXISTING I/O TRANSPORT LIBRARIES

Integrating simulation and data analysis applications into a workflow requires efficient data transport libraries. In this section, we briefly introduce six high performance software packages that we deploy to combine simulation with data analysis applications: 1) MPI-IO, 2) DataSpaces, 3) DIMES, 4) Flexpath, 5) ADIOS, and 6) Decaf.

(1) MPI-IO is a parallel file I/O interface that allows multiple processes of an MPI program to write or read parts of a shared common file [19, 45]. It can map I/O reads and writes to message-passing sends and receives to improve the I/O performance. Unlike the following five software packages, MPI-IO is a low-level I/O library that can support each individual MPI application's file I/O. Also, coupling different applications with MPI-IO requires writing code to let a consumer application know when new data is available in a file.

(2) DataSpaces offers an abstraction of virtual shared space that is distributed across a number of dedicated data servers [8, 11]. It can support data coupling at runtime. In DataSpaces, each participant application is launched by its own *mpirun* or

aprun command such that there are multiple failure domains. If one application fails, the other applications can still survive. DataSpaces provides *put* and *get* functions that use RDMA to write/read data to/from the dedicated data servers. It also provides reader-writer locks to coordinate accesses to shared data among different applications.

(3) DIMES is another data staging library that is provided by the DataSpaces project [8, 50]. Similar to DataSpaces, it supports runtime data coupling, and has multiple failure domains. However, DIMES stores data in RDMA memory buffers located in the simulation application's nodes directly. This way data staging becomes as fast as copying data to main memory. Although *data-storage* servers are not needed, DIMES requires *metadata* servers to manage where data are located and provide locking services to collaborating applications.

(4) Flexpath implements a publisher/subscriber communication mechanism to combine simulations with componentized data analyses [9, 14]. With Flexpath, different software components can be connected by event channels and source-to-sink event communications at runtime. Each publisher or subscriber is executed as an independent application by running *mpirun* or *aprun*. Hence, Flexpath has multiple failure domains. To transfer data, a publisher uses an output epoch (i.e., open, write, close) to save data to its buffer. Later on, a subscriber sends to each of the event publishers a fetch message to request its desired data. Flexpath provides the ADIOS interface (see below) as its own interface.

(5) The Adaptable IO System (ADIOS) supports a range of I/O transport methods [1, 27]. It can be configured to make use of different data-staging libraries such as DataSpaces, DIMES, and Flexpath. In the paper, we call it the "ADIOS/name" transport method if we use ADIOS's interface and use the specific I/O method of *name*. Otherwise, we call it a "native" method for which we use the intrinsic I/O library directly.

(6) Decaf is a dataflow system for parallel communication of participant applications in workflows [13]. It can be regarded as a "coupling service", which allows users to describe nodes and links as serial entities while Decaf takes care of their parallelism. It provides a simple put/get API that utilizes MPI, and can implement a workflow system by using a Python API. Different from the above DataSpaces, DIMES and Flexpath, Decaf creates a single *MPI_Comm_World* for all the participant applications. Data coupling between applications is defined during the compile time. Also, it requires existing MPI-based programs to replace their MPI_COMM_WORLD by the communicator provided by Decaf. Therefore, there is a single failure domain in Decaf workflows.

Next section will compare the performance differences between different workflows that use the above I/O transport libraries.

3 PERFORMANCE ANALYSIS OF WORKFLOWS WITH STATE-OF-THE-ART I/O LIBRARIES

We use the I/O transport libraries of MPI-IO, Flexpath, ADIOS DataSpaces, native DataSpaces, ADIOS DIMES, native DIMES, and Decaf to implement a scientific workflow. The workflow application uses a Lattice Boltzmann method (LBM) based CFD simulation to generate steps of simulation data, which are read and processed by a coupled parallel n-th moment turbulence data analysis application [34, 39]. LBM is a numerical method to solve Navier-Stokes equations and simulate complex fluid flows. It considers fluid as a collection of particles, each of which has random motions [20]. *Collision* and *streaming* are two phases in each simulation time step. *Collision* happens when each particle updates its own distribution function using its local information, and *streaming* happens when a particle exchanges its local information with its neighbors.

We perform workflow experiments on the *Bridges* system from the Pittsburgh Supercomputing Center. *Bridges* has 752 (regular) compute nodes, each of which has two Intel Haswell 3.3 GHz 14-core CPUs and 128GB memory (more detailed system information is provided in Section 6).

Figure 2 shows the various end-to-end time of the CFD workflow experiments using different I/O libraries. Table 1 also presents the experimental setup information of the workflow experiments. On

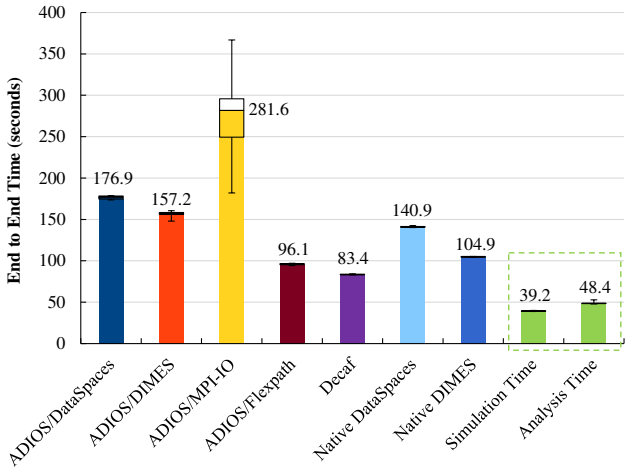


Figure 2: Performance of the CFD workflow application using 7 different I/O transport libraries, in comparison with the simulation time and analysis time.

Table 1: Experimental setup of the CFD workflow experiments shown in Figure 2.

Global input grid size in 3D	$16384 \times 64 \times 256$ ($64 \times 64 \times 256$ per process)
#Simulation processes	256 processes on 16 nodes
#Analysis processes	128 processes on 8 nodes
Compute node information	Each node has 28 cores, 128GB of memory
#Data staging processes	DataSpaces: 32 server processes on 8 nodes DIMES: 32 server processes on 8 nodes Decaf: 64 Decaf-link processes on 8 nodes
#Time steps in the simulation	100, every time step has a data analysis
The n-th moment turbulence data analysis	n=4
Total amount of data moved	400GB

Table 2: Configurations of different I/O transport libraries that have been used to generate Figure 2.

Software Tested	Version	Build Configurations	Runtime Configurations
ADIOS/DataSpaces, and ADIOS/DIMES	DataSpaces: 1.6.2, ADIOS: 1.13	Default ADIOS autoconfig script	lock_type=1, hash_version=2
Native DataSpaces, and Native DIMES	DataSpaces 1.6.2	--with-ib-interface=ib0 --with-dimes-rdma-buffer-size=1024	lock_type=2, hash_version=2
ADIOS/MPI-IO	ADIOS 1.13	Default ADIOS autoconfig script	xml: type="MPI", without *time aggregation*
Flexpath	EVPath, ADIOS 1.13	perl chaos_bootstrap.pl adios-1.13	CMTransport=socket, CM_Interface=ib0
Decaf	https://bitbucket.org/tpeterka1/decaf Git commit version used: 637eb58	mpi_transport=on	redist="count"

Bridges, we build all the software and libraries with gcc 4.8.5 and the Intel MPI library (2017 Update 3). Table 2 particularly lists the software versions and configuration options we use to install and build the tested software systems. Furthermore, we perform large scale experiments using the MPI-IO, Flexpath and Decaf libraries on 13,056 cores as shown in Section 6.3 (see Figures 16 and 18).

Our first attempt tried to use four I/O transport libraries (i.e., ADIOS/{DataSpaces, DIMES, MPI-IO, Flexpath}). Among all the transport methods, MPI-IO performs the worst: it gives the longest and most variational end-to-end time. This is anticipated because MPI-IO writes data to a file system, which is also shared by many other users. However, MPI-IO in the fastest case can still achieve a performance that is comparable to the in-memory methods (e.g., ADIOS/DataSpaces), which had surprised us.

To investigate the problem and enhance the performance of ADIOS/DataSpaces, we turn to the *native* DataSpaces and DIMES libraries. This brings on a significant speedup of 1.3 times for DataSpaces, and a speedup of 1.5 times for DIMES. The reason for the speedup is as follows. ADIOS introduces a uniform interface for all transport methods. However, to achieve this goal, low-level details in certain transport methods have to be hidden in this common interface. For instance, native DataSpaces provides a customized light-weight lock strategy to enforce synchronizations among applications (e.g., dspaces_lock_on_write). The native lock strategy is not exposed by the ADIOS interface. Therefore, we use multiple native DataSpaces locks to implement both *native* DataSpaces and DIMES workflow experiments.

As shown in Figure 2, among all the libraries, Decaf achieves the best end-to-end time of 83.4s, followed by ADIOS/Flexpath of 96.1s. All of our workflow implementations have been designed to overlap simulation with analysis time steps to obtain the best performance. For instance, Figure 3 illustrates how our workflow implementation can hide the analysis time when the simulation time is greater than the analysis time. A similar figure can also be drawn when the analysis time is greater than the simulation time (omitted here). By using such a software design, either the simulation time or the analysis time can be totally hidden from the workflow execution time.

However, the experimental results in Figures 2 show that the workflow execution time is still much larger than the simulation time or analysis time. To investigate why and where the performance is lost, we use TAU [41] and Intel Trace Analyzer and Collector (ITAC [23]) to collect traces of the experiments. Due to the space limit, here we only show the performance analysis results for the three fastest methods (i.e., the native DIMES, Flexpath, and Decaf) to reveal major performance inefficiencies.

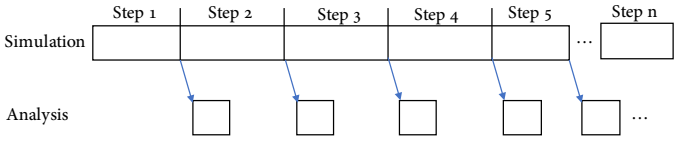


Figure 3: Our workflow implementations can overlap simulation and analysis using I/O transport libraries. In this example, we assume data analysis is faster than simulation for each time step.

Figure 4 shows the trace for the CFD workflow implementation that uses the native DIMES library. In this workflow, simulation needs to synchronize between metadata servers and computing processes, and then inserts results into the DIMES buffer. Notice that there is a lengthy "lock" period, when the simulation is performing data insertions. We use the type-2 customized lock of DIMES, which is a collective lock and enforces strict synchronization between producers and consumers. To better overlap simulation with data analysis, and efficiently utilize the RDMA memory in DIMES, our DIMES workflow uses multiple locks.

The DIMES implementation is presented as follows: we use (*step%num_slots*) as the lock name so that we keep reusing a circular queue of multiple locks with a fixed size of *num_slots*, where *step* is the time step index of the CFD simulation, and *num_slots* is the number of slots the CFD simulation can use to buffer its output data in a FIFO manner. When the analysis application is slower, the simulation application will be stalled in order to make sure the previous data are not overwritten. This scenario is shown in Figure 4 where the application stall time is almost equal to one step of simulation time. As a result, the end-to-end workflow time nearly doubles. [1]

Next, we present the TAU trace for the Flexpath-based workflow implementation in Figure 5. In the figure, we display a snapshot of length of three seconds for two different cases: 1) running simulation alone, and 2) running the Flexpath workflow. The orange stripes represent the time to execute the MPI_Sendrecv function, which performs the inter-process communication in the streaming phase of the LBM simulation. We can see that after adding the

[1]Workflow implementations with DIMES can be further optimized by using an additional thread in the consumer application to fetch newer version of data while the main thread is analyzing the data of previous time steps.

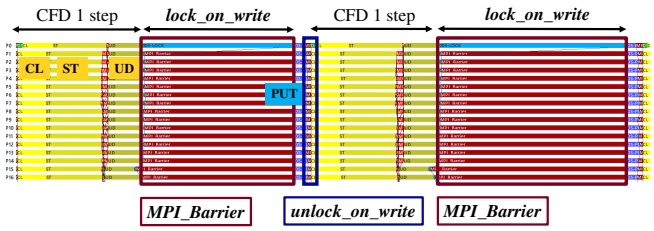


Figure 4: A trace of native DIMES with a snapshot of 2 seconds.

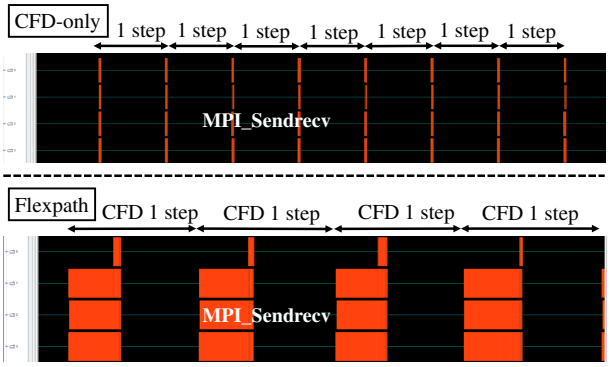


Figure 5: Comparison between running CFD simulations only and running Flexpath based workflows. This figure shows a snapshot of 3 seconds.

Flexpath data staging, the MPI_Sendrecv time in the LBM simulation takes much longer, which results in increased end-to-end time. Because both LBM's streaming operation and Flexpath's event channel involve intensive communications, Flexpath's data-staging operations will compete with the simulation's MPI communication. In particular, when staging a large slab of simulation data (e.g., 16 MB per time step per process in this workflow experiment), the chances to have communication interferences are much higher.

Finally, we compare the fastest workflow implementation that uses the Decaf method (whose performance is shown in Figure 2) to the experiment that runs simulation only. We are not able to use TAU for the tracing purpose, because the latest TAU library (version 2.27) cannot filter out the huge number of inline Boost serialization function calls made by Decaf. The inline function calls make the trace files too large to generate. We have reported the problem to TAU developers, and they are working on it. To circumvent the tracing problem, we manually instrument the workflow source code, and use the Intel Trace Analyzer and Collector (ITAC) software to collect execution traces.

Figure 6 shows the two traces for CFD simulation only, and Decaf-based workflow, respectively. In the CFD simulation only trace, each time step contains three major computation kernels: collision (CL), streaming (ST), and update (UD). In a trace snapshot for 0.9 seconds, CFD simulation itself can execute 3 time steps. Note that all time steps have the similar performance pattern. By contrast, in the lower Decaf-based workflow trace, there is an additional PUT function invoked by simulation processes to transfer output data to

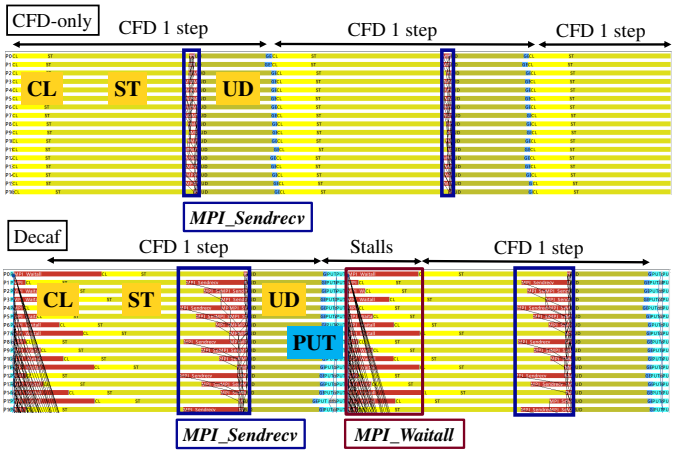


Figure 6: Comparison between running CFD simulations only and running Decaf-based workflows. This figure shows a snapshot of 0.9 seconds.

link nodes via Decaf. We observe that the PUT function utilizes a collective "MPI_Waitall" function, during which time all simulation processes stall. This is because Decaf has to make sure data is safely stored in the link nodes before it can proceed to the next step. We also observe that the "MPI_Sendrecv" time (within the streaming ST phase) increases significantly after Decaf is added. This indicates that using Decaf has affected the MPI communication performance of the original simulation application.

4 THE ZIPPER RUNTIME SYSTEM AND IMPLEMENTATION

From the above performance analysis, we find multiple performance issues and optimization opportunities as follows: 1) The staging-server access cost including the server query, data movement and locking service can be reduced (e.g., DataSpaces and DIMES have such a cost); 2) the enforced global barriers for all writer processes and all reader processes can be reduced (e.g., Decaf and Flexpath have such barriers); 3) the data transfer time between consecutive simulation steps can be hidden by computation time, and decreased by an early-start fine-grain pipelining approach (e.g., we will increase the degree of task-level parallelism and use pipelining to overlap all simulation, analysis, and I/O tasks); and 4) asynchronous fine-grain-block data transfers have a more balanced network traffic, which can have a less interference with the original application's communication time than a burst of large data block transfers (e.g., Decaf and Flexpath have experienced increased MPI communication time in the original simulation application).

The rest of this section will introduce a new runtime system called *Zipper* to improve the above identified performance inefficiencies.

4.1 System Overview

In our system design, both simulation and analysis applications are executed in parallel using different compute nodes of an HPC system. For instance, we allocate m compute nodes to execute the

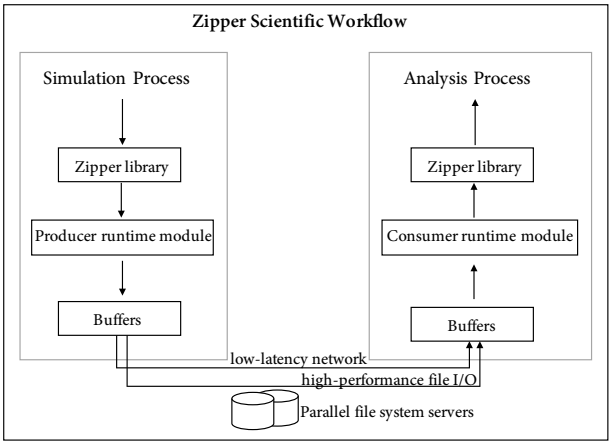


Figure 7: Architecture of the Zipper workflow framework to integrate a parallel simulation application with a parallel analysis application.

simulation application, and allocate *n* compute nodes to execute the data analysis application simultaneously. The analysis application is driven by data-availability. At the same time, the simulation application pushes data to the analysis application continuously (i.e., using two-sided data transfers). Whenever a new data block arrives, the analysis application will immediately read and process it.

The architecture of the Zipper system is shown in Figure 7. Both simulation process and analysis process use the *Zipper library* to output or input data, respectively. The interface provided by the Zipper library is simple: Zipper.write(block_id, void* data, block_size) and Zipper.read(block_id, void* data, block_size). The Zipper.write() method passes data to the *Producer Runtime Module*. The producer runtime module is multi-threaded and provides the essential functionalities of buffer management, asynchronous I/O, data prefetching, communication with consumers, and the concurrent data transport optimization. On the other side, the analysis application works as a consumer. Each analysis process uses Zipper.read() to interact with its *Consumer Runtime Module* to get data constantly. Both producer and consumer runtime modules can utilize low-latency HPC network and high-performance parallel file system to transport and store computed results.

The Zipper system offers two modes to users: *Preserve* mode and *No Preserve* mode. A user may choose the *Preserve* mode to keep the computed results for future analysis, validation, and verification. On the other hand, one may choose the *No Preserve* mode to save storage space and perform faster experiments.

4.2 Implementation

Figure 8 shows the producer runtime module. It consists of a producer buffer, a sender thread, and a writer thread. The sender thread is responsible for sending data to the consumer processes via the HPC network. The writer thread is responsible for storing computed results to a parallel file system. More specifically, the sender thread checks whether there are blocks stored on disks, and then appends the on-disk block IDs to form a *mixed message*. Notice that

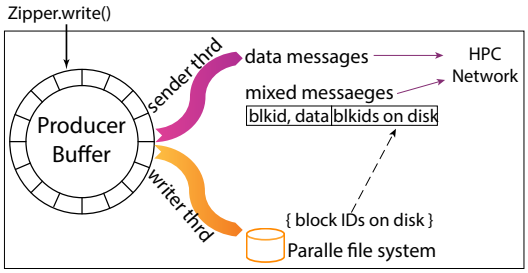


Figure 8: The producer runtime module.

even when the analysis application is slower than the simulation, the simulation application will not be blocked or stalled since the writer thread is also moving data to the parallel file system. In Subsection 4.3, we will describe how the writer thread can help the sender thread to increase data transfer rate by using a concurrent dual data-path method.

Figure 9 shows the consumer runtime module, which consists of a consumer buffer, a receiver thread, a reader thread, and an output thread. The receiver thread gets a mixed message from the HPC network, and divides it into a data block and a list of block IDs. The data block will be moved to the consumer buffer and the block IDs will be copied to an array of "*block IDs on disk*". The reader thread will read the block from the parallel file system and put it to the consumer buffer. The data block itself contains all the necessary information that the analysis application will need, which includes the time step index, the process ID that sends the block, and the position of the data block in the global input domain. This way the consumer process knows which specific block it receives and can apply appropriate data analysis to it.

The output thread in Figure 9 is dedicated to supporting the *Preserve* mode. It constantly fetches data blocks from the consumer buffer. If the fetched data block has a flag of *on_disk = false*, the output thread will store the data block to the file system. A data block in the consumer buffer can be freed from the system only if the block has been both analyzed by the analysis process and stored to the file system by the output thread. Note that the output thread will not be created by the runtime system in the *No Preserve* mode.

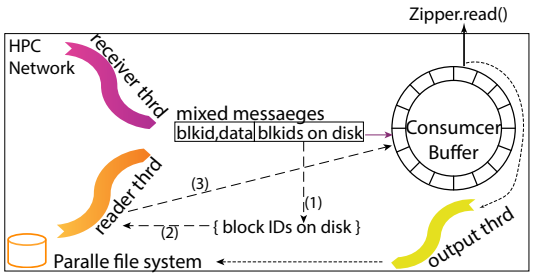


Figure 9: The consumer runtime module.

4.3 Optimization of Concurrent Message and File Data Transfers

The Zipper runtime system relies on two data paths to transport data: 1) message passing via a low-latency HPC network, and 2) parallel I/O via a parallel file system. We use the parallel file system because we need it to alleviate the simulation stall time when the analysis application is relatively slow such that the simulation application is blocked.

On the other hand, using two data paths has the potential to increase the data transfer rate if a portion of the data movement work is offloaded to parallel file I/O. Figure 10 explains how a concurrent transfer optimization may work. The top part shows that all data blocks are sent by network. The bottom part shows that most blocks are transferred by network while a few blocks are transferred by parallel file I/O. Considering that emerging HPC systems will deploy much faster non-volatile memory (NVM) technologies, future HPC systems will benefit more from this optimization.

Our *concurrent data transfer optimization method* is implemented as a *work-stealing* algorithm, which allows data blocks to be sent through the parallel file system path only when it is necessary. The writer thread in the producer runtime module works like a helper. When detecting the producer buffer is almost full (defined by a "high water mark" threshold), the writer thread will fetch a data block from the buffer and send it to the file system. Algorithm 1 shows the pseudocode of the adaptive writer thread. This strategy can automatically adapt to either the message-passing-only method or the mixed network&file-IO method depending on how full or empty the producer buffer is. For instance, if the buffer is constantly near-empty, Zipper will always use the fastest HPC network to send data to the analysis application (Section 6 shows the experiments and effect of using the concurrent data transfer optimization).

We use hardware performance counters to monitor network traffic and verify the cause of the speedup by using the concurrent data transfer optimization method. If an HPC system has two separate networks (i.e., one for message passing and the other for I/O traffic), we expect the proposed concurrent data transfer optimization will increase the data transfer rate. If an HPC system does not have a segregation of communication traffic and I/O traffic (such as the Bridges system and the Stampede2 system used in Section 6), the concurrent data transfer optimization may not be able to reach its highest potential. Nevertheless, we still observe a significant speedup on Bridges and Stampede2 (detailed experiments are shown in Section 6). Here, we briefly introduce the reason. Since both InfiniBand and Omni Path Architecture (OPA) networks have

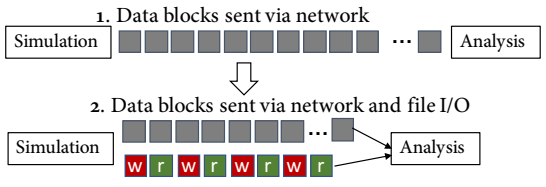


Figure 10: The concurrent data transfer method can reduce the data transfer time by converting a portion of message passing time to certain overlapped parallel file I/O time.

Algorithm 1 Writer Thread Work-stealing Algorithm

```
1:  while true do
2:      block ← StealBlock(ProducerBuffer)
3:      store the block to the parallel file system
4:      place the block's ID into the in-memory data structure of
        block IDs on disk
5:  end

6:  function StealBlock(ProducerBuffer)
7:      while true do
8:          acquire the lock of ProducerBuffer
9:          if #Blocks in ProducerBuffer > Threshold then
10:             fetch the address of the first block in ProducerBuffer
11:             release the lock of ProducerBuffer
12:             return the address of the block
13:         else
14:             wait on a condition variable and release the lock
15:             /* Note: the computation thread will produce data and signal the
    condition variable when #Blocks in ProducerBuffer > Threshold. */
```

network congestion control mechanisms, when many simulation processes try to send data to many analysis processes simultaneously, network congestion control in network switches will play a key role in performance. Our concurrent data transfer optimization method is more efficient in working with the congestion control mechanism because our dual paths allow messages (i.e., the data blocks) to arrive out of order and take different network paths, to ease network congestion and take advantage of multiple network links/switches for improved bandwidth. In-depth performance analysis will be presented in Subsection 6.2.

Brief summary of Zipper's features: In summary, 1) Zipper uses fine-grain data blocks and creates a higher degree of task parallelism to accelerate the pipeline execution. The other in-situ workflow systems often generate one big data block per time step. 2) Zipper does not impose strict barriers between time steps, and deploys a dataflow-driven approach to minimizing application stalls. The other workflow systems often force using strict writer-reader interlocks and collective global operations (e.g., wait_wall, global locks). 3) There is no server overhead involved, which is different form DataSpaces and DIMES. 4) Zipper supports multiple failure domains (similar to DataSpaces, DIMES, and Flexpath). And 5) Zipper supports both Preserve mode and No-Preserve mode, and introduces a concurrent data transfer optimization, which is based on an adaptive work-stealing algorithm.

4.4 Performance Model

To evaluate the efficiency of Zipper, we use a simplified analytical performance model to estimate the workflow end-to-end time. The analytical model uses the following notation. A number of P processor cores are used to compute simulation, and a number of Q processor cores are used to analyze results. The total amount of simulation data generated is D. Given a fine-grain data block of size B, there would be $n_b = \frac{D}{B}$ blocks. In the experiments, we use block sizes that are between 1MB and 8MB.

To keep our analytical model simple, we assume that each simulation processor core computes $\frac{n_b}{P}$ blocks, and each analysis processor core analyzes $\frac{n_b}{Q}$ blocks. Nevertheless, the model can be adapted to support load imbalance situations by considering the process with the maximum workload. The analytical model is based upon the time spent on each data block. Since we use the pipelining parallelism to couple applications, a data block will go through different stages: Simulation → Transfer result → Analyze result.

Let t_c, t_m, and t_a denote the time to compute a data block, transfer a block, and analyze a block, respectively. We model the parallel computation time T_{comp} as $t_c \times \frac{n_b}{P}$, and model the parallel analysis time T_{analysis} as $t_a \times \frac{n_b}{Q}$. Because each pipeline stage works independently from any other stage, the end-to-end time-to-solution can be expressed as follows: $T_{t2s} = \max(T_{\text{comp}}, T_{\text{transfer}}, T_{\text{analysis}})$. We assume that the number of data blocks is much greater than the number of pipeline stages for which we can ignore the pipeline *startup* time and *drainage* time. In the paper, we use the analytical model to show the end-to-end time is almost equal to the time of one stage. A more detailed model that can accurately predict performance would require modeling the time to compute a block, transfer a block, and analyze a block (for any data block size from small to large), as well as network contention/congestion given a block size and different numbers of P and Q. Our future work will study how to build a more detailed performance model.

The simplified T_{t2s} formula can be easily derived from a pipeline diagram. For instance, as shown in Figure 11, different stages are overlapped such that the end-to-end time is almost equal to the time of the slowest stage. Based upon the model, if the simulation application and analysis application are scalable, the Zipper workflow can scale well accordingly.

Note that the data transfer time of T_{transfer} can be controlled by the frequency to output the simulation data (e.g., one data output per k time steps) to reduce the I/O time. In Section 6, we will perform a variety of experiments to verify the model.

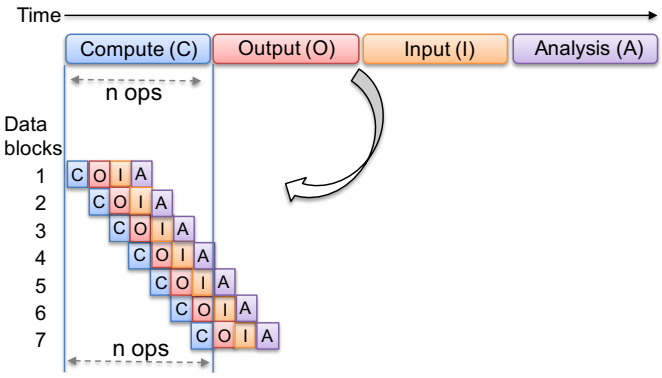


Figure 11: Non-integrated design (upper) vs. integrated design (lower). In the (lower) integrated design, at any time, four stages (C, O, I, and A) are working on four distinct data blocks. The four data blocks could be sequentially dependent, but can still be processed in parallel due to the data pipelining parallelism.

5 RELATED WORK

In the conventional *post data processing* methods [21, 32, 42], a simulation application computes and stores computed results to files. Next, an analysis application is launched to perform various data analyses. Due to the post-processing methods' expensive I/O cost, *in-situ* approaches are introduced to analyze data when the data are still in memory [4, 5]. For instance, Paraview/Catalyst [15] and VisIt/Libsim [47] can be used to perform in-situ analysis and visualization on large datasets in memory. Paraview/Catalyst defines an interface between simulation and visualization applications, which requires developers implement three subroutines: initialize, coprocess, and finalize. Similar functions are also provided by VisIt/LibSim to support in-situ visualization.

As an alternative to in-situ approaches, *data staging* approaches can enable co-analysis pipelines by using a loosely coupled integration model. ADIOS [1], PreDatA [51], GLEAN [46], DataStager [2], DataSpaces [11], DIMES [50], and Flexpath [9] leverage advanced I/O infrastructure to reduce the I/O cost. In particular, PreDatA [51] realizes in-transit data processing along a data flow. It moves data from compute nodes to staging nodes through two passes: the first pass of sending data-fetch requests to the staging nodes, followed by the second pass of *pulling* packed data chunks from the compute nodes. We use a single pass to move data to the analysis processes rapidly. DataSpaces [11] and DIMES [50] allow different applications to store data to and extract data from dedicated servers (or metadata servers) simultaneously. Our Zipper system does not use dedicated servers and has no accompanying server access overhead. Sun et al. [43] use DataSpaces and asynchronous coupling of workflows as a user case to develop scheduling polices for placing data to different staging cores. GLEAN [46] and DataStager [2] deploy a data staging service on analysis nodes of a cluster to support in-situ processing. FlexIO [52] uses local memory and RDMA to support co-analysis either on the same compute nodes or on different staging nodes. Our research shares the data-staging philosophy of theses libraries (e.g., data coupling at runtime and multiple failure domains), but uses fine-grain data blocks, asynchronous task parallelism, and holistic end-to-end level pipelining to minimize application idle time, reduce network contention, and overlap all workflow stages (i.e., simulation, data write, data read, and data analysis).

Our concurrent data transfer optimization method improves the communication throughput by taking advantage of the network congestion control and multiple switches and links. Our deployed network congestion measurement is inspired by the work of Alali et al. [3], which conducts a study to understand whether network congestion occurs on production HPC systems. There are also studies that investigate how to use Quality of Service (QoS) mechanisms to enhance communication. Reinemo et al. compare a list of QoS capabilities on InfiniBand, Advanced Switching, and Ethernet [38]. Gonsiorowski et al. create a model to analyze the use of QoS lanes to reduce the impact of the RAID *rebuild* traffic by assigning different traffic quotas to read, write, and rebuild operations. [18]. Kim et al. design an OpenSM (Open SubnetManager) based scheme to adjust the QoS level dynamically by considering the estimated bandwidth and requirement to increase the overall bandwidth of multiple concurrent traffics [25].

Workflow systems such as Pegasus [10], Kepler [28], Taverna [48], and Condor/DAGMan [24] use files to communicate data and target coarse job-level meta-scheduling. Decaf [13] is a workflow middleware that uses multiple overlapping MPI communicators and a special staging area called "link" to transfer data between a producer and a consumer. The communication among Decaf producer, link, and consumer are inter-locked, and all data must arrive in link before they can be forwarded to the next application. Also, slower consumers will block the producers from running. Swift/T [49] uses a Swift-Turbine compiler to translate a Swift program to an ADLB [30] MPI program, and executes it with a master-worker model. Differently, we target fine-grain tasks and asynchronous computing, and use data-staging to minimize the workflow latency.

6 PERFORMANCE EVALUATION

This section evaluates the performance model, concurrent message and file transfer optimization, and scalability of the Zipper system on two different supercomputers: *Bridges* and *Stampede2*.

The *Bridges* system from the Pittsburgh Supercomputer Center (briefly mentioned in Section 3) has 752 regular nodes (128GB memory each), 42 large shared-memory nodes (3TB memory each), and 4 extreme shared-memory nodes (12TB memory each). Each node has 28 Intel Haswell cores. *Bridges* deploys a 100 Gbps Intel Omni-Path Architecture, which connects all compute nodes with a 10PB high performance Lustre parallel file system.

The *Stampede2* system in the Texas Advanced Computing Center entered full production in August 2017. It has 4,200 Knights Landing nodes. Each node has a self-booting Knights Landing (KNL) processor (68 cores), 96GB of DDR memory, and 16GB of MCDRAM (Multichannel DRAM), and peak performance of 3 Teraflops per node. *Stampede2* uses an Intel Omni-Path Architecture and has a 30PB Lustre parallel file system.

We perform experiments with three synthetic applications and two real-world scientific computing applications. Information of the applications is presented in Table 3.

6.1 Evaluation of the Performance Model

We describe an analytical performance model in Section 4.4 showing that the Zipper system ideally should obtain end-to-end time of $T = \max(T_{\text{comp}}, T_{\text{transfer}}, T_{\text{analysis}})$. Our first experiment is intended to verify whether the performance model conforms to the actual Zipper workflow's performance. The experiments were performed on *Bridges* using 1,568 CPU cores for simulation and 784 CPU cores for data analysis in both *No Preserve* and *Preserve* modes. In the experiments, a total amount of 3,136GB of data are transferred from simulation to analysis.

Figure 12 shows the *No Preserve* mode's time breakdown for three synthetic applications (i.e., the $O(n)$, $O(n \log n)$, and $O(n^{3/2})$ applications listed in Table 3) using two block sizes of 1MB and 8MB. In the synthetic workflow, each data block is analyzed and its standard variance is reduced to one double-precision floating point value. For each block size (i.e., 1MB and 8MB), we show the measured simulation time, data transfer time, and analysis time, as well as the workflow's end-to-end time.

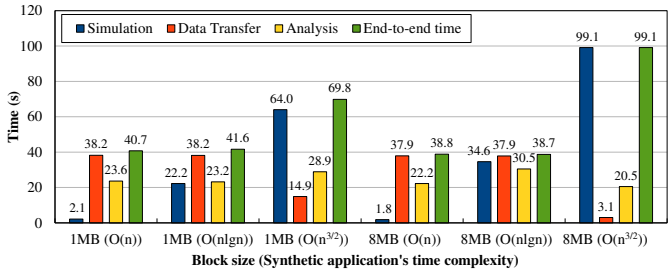


Figure 12: Time breakdown of the execution time for three different synthetic applications in the *No Preserve* mode.

As depicted in the figure, given the same block size, as the application's time complexity T(n) increases, the dominant stage switches from data transfer time (in red color) to simulation time (in blue color). However, regardless of the distinct synthetic applications, the workflow's end-to-end time is always close to the maximum stage time, which empirically validates our performance model.

Next, we do the same experiments using the *Preserve* mode. Figure 13 shows the corresponding time breakdown and total time. The experiments show that the end-to-end workflow time is almost equal to the time spent on storing computed results to the file system. Since all processes have generated a total amount of 3,136 GB of data, storing data to disks takes the longest time.

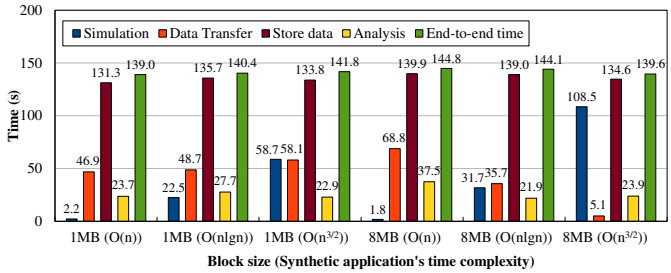


Figure 13: Time breakdown of the execution time for three different synthetic applications in the *Preserve* mode.

Moreover, we evaluate the performance model with two real-world applications of CFD and LAMMPS. Their results are shown together with the weak-scalability experiment (in Subsection 6.3). For the CFD and LAMMPS applications, their workflow end-to-end time is nearly the same as the dominant simulation time.

6.2 Effect of the Concurrent Message and File Transfer Optimization

Our second experiment will evaluate the effect of using the concurrent message and file data transfer optimization.

The three synthetic applications in Table 3 are used to do experiments on *Bridges*. We instrument the applications' source code, and measure the time spent on two parallel threads of each simulation process: the computation thread, and the sender thread. The computation thread will be either computing simulations or stalled due to a full producer buffer (represented as stacked *simulation* and *stall* in Figure 14). Similarly, the sender thread will be either

Table 3: Description of the applications used in our experiments.

Workflow applications	Simulation	Data analysis
Synthetic $O(n)$	To emulate T(n)=$O(n)$ linear algorithms	Standard variance computation
Synthetic $O(n \log n)$	To emulate T(n)=$O(n \log n)$ such as divide&conquer algorithms	Standard variance computation
Synthetic $O(n^{3/2})$	To emulate T(n)=$O(n^{3/2})$ algorithms such as matrix computations	Standard variance computation
CFD application	Use the Lattice Boltzmann method to compute 3D channel flows	Turbulence analysis
LAMMPS application	Use LAMMPS to compute 3D Lennard-Jones atoms melt dynamics	Atoms movement statistics

sending messages or waiting for new data (represented as stacked *data transfer* and *stall*).

As seen in Figure 14, we increase the number of CPU cores from 84 to 2,352 to perform weak scaling experiments. For each specific number of cores, we compare the implementation that uses the message-passing-only method to the implementation that uses the concurrent message&file transfer optimization. Given n cores, there is a group of four columns in the figure. The left two columns show the performance of the message-passing-only implementation, and the right two columns show the performance of the concurrent transfer optimization.

In Figure 14.a for the $O(n)$ application, from 84 to 2352 cores, the simulation application's wallclock time has been reduced by 32.4%, 26.3%, 29.2%, 16.1%, 29.4% and 20.2%, respectively. This improvement is mainly due to the reduced stall time. For this $O(n)$ application, the data generation rate from each compute node is 56GB/s, while the point-to-point network bandwidth for each port is 10.2GB/s. As a result, the sender thread cannot move data out in time, and the producer buffer becomes full and the simulation thread is blocked. In this case, our work-stealing writer detects that the threshold is reached and starts to steal blocks (stolen 47%∼62.4% of total blocks) in the above cases.

In Figure 14.b for the $O(n \log n)$ application, the concurrent transfer optimization has reduced the simulation stall time and data transfer time by 8.1%, 14.2%, 21.7%, and 22.5%, from 336 to 2352 cores, respectively. Our work stealing doesn't improve the two smaller cases of 84 and 168 cores because the producer buffer is mostly empty and there is nothing to steal during the execution.

Figure 14.c shows the time for the computation-intensive $O(n^{3/2})$ application. Since this application has the slowest data generation rate, the producer buffer is almost always empty such that the work-stealing in the writer thread is never activated. In this case, the concurrent transfer optimization falls back to the message-passing method.

Based on the performance results in Figure 14, we can find that the concurrent optimization method is always as good or better than the message-passing-only method. The reason is that the concurrent optimization deploys an adaptive stealing-based approach such that it lends a hand only if there exist appropriate opportunities to steal. If there is no stealing opportunity, its performance will be the same as the original performance.

6.2.1 Why the concurrent optimization can improve performance. The HPC system of *Bridges* uses the Intel Omni Path Architecture (OPA) network, where each compute node is connected to a leaf edge switch (42 ports, 12.5 GB/s each) and then all leaf switches are connected through a set of core edge switches [6]. At first glance, it seems to be impossible to gain any benefits by using the concurrent transfer optimization because there is only one link from a compute node to one port of a leaf switch.

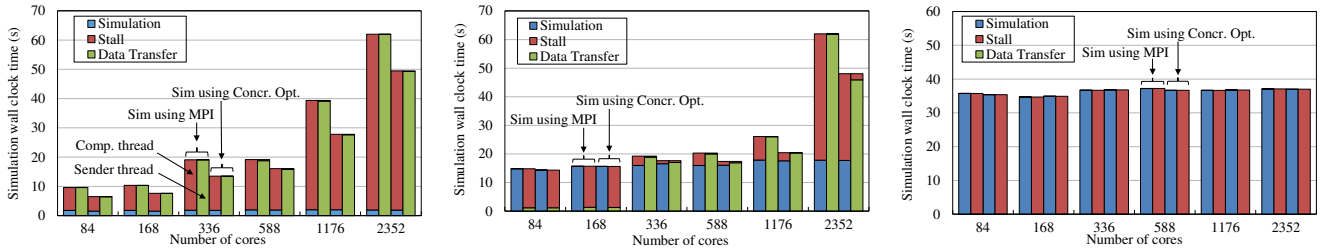
To dig into the reason, we use the PAPI network component [36] and OPA network analysis tools to measure network related performance events. We measure the performance counters of `XmitData`, `XmitPkts`, `RcvData`, `RcvPkts`, and `XmitWait` when we compare the message-passing only method and the concurrent method. Since users do not have privileges to access the counters on switches, we can only collect the performance counters on the network adapter on each compute node.

Among all the network events, we find that the `XmitWait` counter shows the biggest difference between using the message-passing only method and using the concurrent method. The specific `XmitWait` counter is used to count the number of events (in FLIT[2]) when any virtual lane had data but was unable to transmit [22], for reasons such as no transmission credits available, or the link was busy sending non-data packets. Hence, this counter is often used to measure the extent of network congestion [3].

We use the Linux command "`opapmaquery -o getportstatus`" to collect the values of the counters on each compute node periodically. Whenever 10% of the total number of blocks are generated, our sender thread will query the counters and calculate the difference between the current query and the previous query. This measured difference indicates how many messages are attempted to send out but rejected due to the network congestion control mechanism. The larger the `XmitWait` value is, the more times the network adapter is unable to transmit, and the more congested the network is.

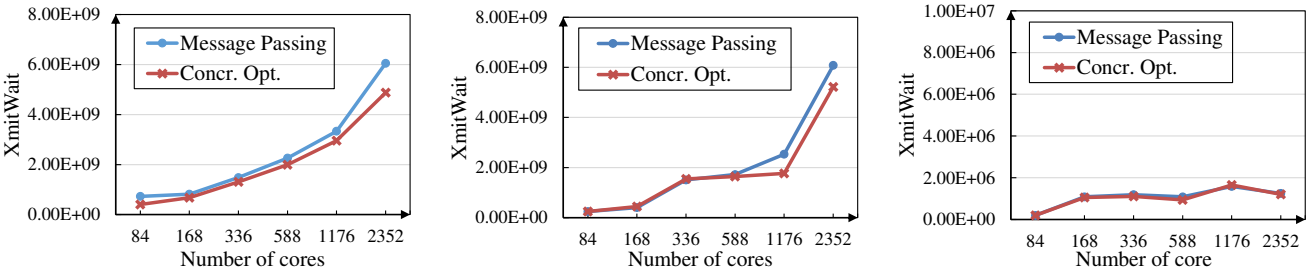
We use the measured `XmitWait` counter to show the relationship between the degree of the network congestion and the data transfer time. As shown in Figure 15.a dedicated for the $O(n)$ application, we observe that the counter of `XmitWait` using message-passing-only is larger than that using the concurrent method by 80%, 21%, 13%, 13%, 13%, and 24% from 84 to 2,352 cores, respectively. This suggests that when we use the message-passing-only method, more messages are not able to transmit than when we use the concurrent method. Since `XmitWait` is an indication of the degree of network congestion, we can say that the concurrent method has less serious congestion than the message-passing-only method. Also due to the reduced network congestion, the concurrent method can send data more quickly and has shorter transfer time, which is confirmed by Figure 14.a correspondingly.

[2]In Omni Path, the Link Transfer (LT) layer segments the end-to-end Fabric Packets (FPs) into 64 bit Flow Control Digits (FLITs), and groups 16 FLITs into a Link Transfer Packet (LTP) to reliably transport FP FLITs and control information on the link[6].



(a) $O(n)$ application. (b) $O(n \log n)$ application. (c) $O(n^{3/2})$ application.

Figure 14: Effect of the concurrent data transfer optimization using different number of cores on three synthetic applications.



(a) $O(n)$ application. (b) $O(n \log n)$ application. (c) $O(n^{3/2})$ application.

Figure 15: Network Congestion of the concurrent data transfer optimization using different number of cores on three synthetic applications. *XmitWait* **counts the number of occurrences when any virtual lane had data but was unable to transmit.**

Measurement of the `XmitWait` counter for the $O(n \log n)$ application is shown in Figure 15.b. On 84 and 168 cores, the `XmitWait` counter is less than 0.5×10^9, which implies a light network congestion and all data can be sent out rapidly without waiting. The other sign of a light network congestion is that the producer's message buffer is almost empty all the time. Therefore, our writer thread does not steal any data blocks such that the concurrent method becomes the message-passing-only method. Hence, Figure 14.b shows equal data transfer time on 84 and 168 cores. However, for larger scales starting from 336 cores, the `XmitWait` counter rises up significantly (i.e., 3 times to 12 times bigger than that on 168 cores). This suggests a higher degree of congestion, and the producer's buffer becomes full and the writer thread starts stealing and eases the congestion again. The reduced congestion also justifies the shorter data transfer time by using the concurrent method from 588 to 2352 cores (see Figure 14.b).

In Figure 15.c, for the slowest $O(n^{3/2})$ producer application, the value of the `XmitWait` counter is around 10^6 (i.e., three orders of magnitude less than the previous two applications). The congestion degree is constantly low for all different numbers of cores, and the producer's buffer is almost empty such that the concurrent method falls back to the message-passing-only method. Therefore, the corresponding Figure 14.c shows that the message-passing-only and concurrent methods have equal data transfer time.

6.3 Scalability Performance

The last experiment is to evaluate the scalability performance of the Zipper system. We perform experiment with two real-world applications of CFD and LAMMPS on the larger *Stampede2* system. *Bridges* only allows 4,704 cores per job.

The CFD application uses the Lattice Boltzmann method to compute 3-D simulations of viscous incompressible fluid sliding down 3D hydrophobic microchannel walls [20, 53]. Its corresponding analysis component computes the n-th moment of the velocity distribution: $E(u(x,t)^n)$, where $u(x,t)$ is the velocity at a spatial point x at time t. The statistics can help scientists understand the properties of the turbulent flow with high *Reynolds* numbers. When all n-th moments are available, the probability density function of $u(x,t)$ can be evaluated to give the complete information of the velocity fluctuation of a turbulent flow [29, 40].

The LAMMPS application simulates clusters of Lennard-Jones atoms. We use the application to study the melting process of materials from a low-energy solid structure at low temperatures to a set of higher energy liquid structures at high temperatures. The Lennard-Jones model is a mathematical model for approximating interactions between neutral atoms or molecules. The counterpart data analysis application will compute MSD (mean squared displacement). MSD calculates the deviation time between the position of a particle and a reference position, in order to analyze the spatial extent of random motions.

Remark: The reason we select the CFD and LAMMPS workflows to do experiments is that *simulation-time data analyses* are common in scientific and engineering domains, and achieving high performance is crucial to most domain scientists [12, 26, 33]. The data analysis application in our workflows receives data blocks and analyzes them accordingly, followed by asynchronous reduction

operations. Our future work will add a simplified programming interface (e.g., an application interface similar to MapReduce) to Zipper to simplify parallel programing of big data analysis.

6.3.1 The CFD application. In the CFD workflow experiments, each simulation process is allocated with a fluid subgrid of dimension 64×64×256. When doubling the number of CPU cores, the total input size also doubles (i.e., weak scaling). Among the total number of cores, two thirds of the cores are used for CFD simulations and one third are used for the n-th moment analysis.

Figure 16 shows the end-to-end time using MPI-IO, Flexpath, Decaf, and Zipper, as well as the simulation-only time in the *No Preserve* mode. On Stampede2, when the number of compute nodes is larger than 8, DataSpaces and DIMES aborted with "rpc_bind_addr" error in the DataSpaces/DIMES initialization function. The error is related to "an issue related to OPA and KNL processors", and has been confirmed by the DataSpaces team. Hence, we could not test DataSpaces/DIMES on Stampede2. Nevertheless, the fastest library is Decaf, which we choose to compare with Zipper.

Simulation-only time is the time spent only by the simulation program's computational kernels (excluding any I/O, idle time, and data staging related cost). It works as a lower bound of the workflow end-to-end time. As depicted in Figure 16, we can see that using MPI-IO is not scalable: as the number of cores increases from 3264 to 13,056, larger MPI-IO experiments take too long to finish. On the other hand, Flexpath and Decaf scale well from 204 cores to 3,264 cores. However, Flexpath and Decaf crashed with software faults on 6,528 and 13,056 cores. In particular, Decaf has segmentation faults due to integer overflows. We have reported the issue to Decaf developers and they have confirmed the error. Flexpath terminated with segmentation fault when the number of cores reaches 6,528. We have also reported the problem to Flexpath developers.

In order to show complete experimental results for Flexpath and Decaf, we assume that both methods have perfect scalability on 6,528 and 13,056 cores, and show their ideal end-to-end time (denoted by dotted lines). As shown in Figure 16, Zipper's end-to-end time is almost equal to the simulation-only time, and is 11.5 times faster than Flexpath, and 1.7 times faster than Decaf.

One might wonder why Flexpath is slow. We conducted a set of investigations to find out the reason. Based on our experiments, Flexpath's data transfer time becomes significantly slower as we increase the number of processes per node (each process uses Flexpath to transport data). Our finding is that Flexpath does not have optimized support for multiple processes per node. Flexpath utilizes a socket interface and all communications (even within the same node) have to go through the socket interface. However, the communication between processes on one node can use shared memory to achieve higher performance (e.g., MPI uses this optimization). In order to show the *ideal* performance of Flexpath, we attempt one-process-per-node to rerun the 204-core experiment (although wasting many cores on each node). In the new experiment, Flexpath using 102 processes on 102 nodes (i.e., 6,936 cores) only takes 46 seconds, but is still slower than Zipper using 102 processes on 3 nodes (i.e., 204 cores) by 16.8%. Besides using a smaller number of processes per node, another Flexpath optimization is to use a "Master" process on each node to aggregate data from all processes of the node to reduce the communication cost. However, this method requires significant code modifications.

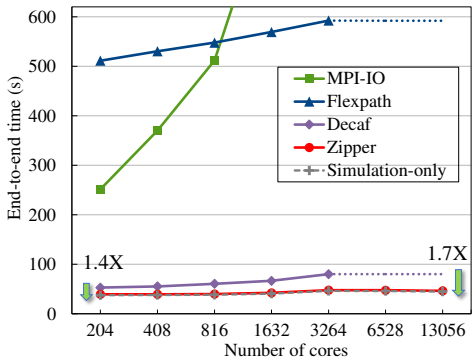


Figure 16: Scalability performance of the CFD workflows using MPI-IO, Flexpath, Decaf and Zipper, respectively.

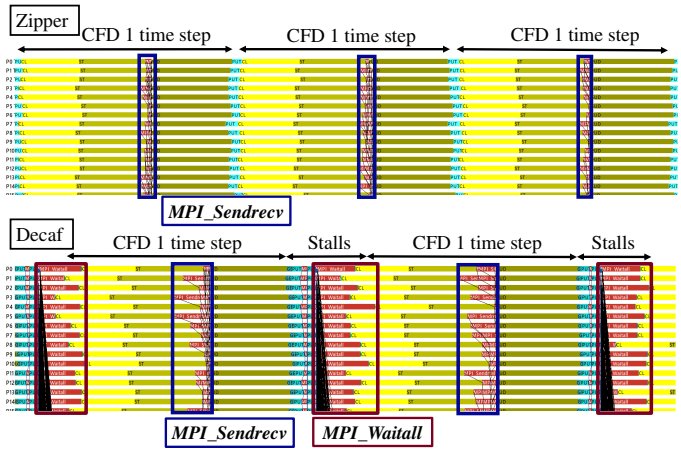


Figure 17: Trace comparison between Zipper and Decaf for the CFD application on 204 cores. This figure shows a snapshot of 1.3 seconds when using 204 cores, which is taken from the experiment shown in Figure 16.

In order to illustrate why Zipper is faster than Decaf, Figure 17 displays Zipper and Decaf's traces within a time interval of 1.3 seconds on 204 cores. To take the snapshot, we zoom in the entire trace, and then cut out a trace segment of 1.3 seconds. Note that showing the entire trace all at once will make the figure too dense to view any details. During the same interval, Zipper is able to run three simulation steps, while Decaf is able to run two steps with a significant amount of stall time. This speedup of 1.4 times is almost the same as the speedup shown in Figure 16 on 204 cores.

The reason for the performance inefficiency is as follows (also reported in Section 3): 1) Decaf has significant simulation stall time caused by MPI_Waitall, and 2) the simulations application's MPI_Sendrecv time becomes longer due to Decaf's interference. Since Zipper uses smaller data blocks and asynchronous pipelining data transfers, both the network traffic interference and the collective MPI cost have been reduced.

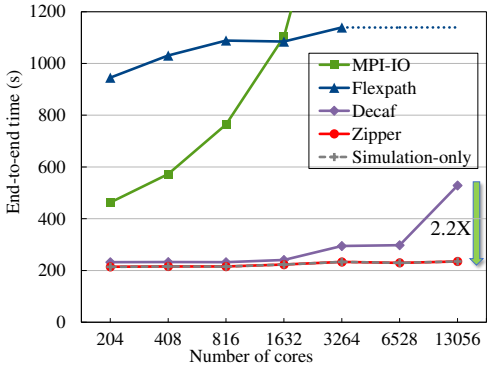


Figure 18: Scalability performance of the LAMMPS workflows using MPI-IO, Flexpath, Decaf and Zipper, respectively.

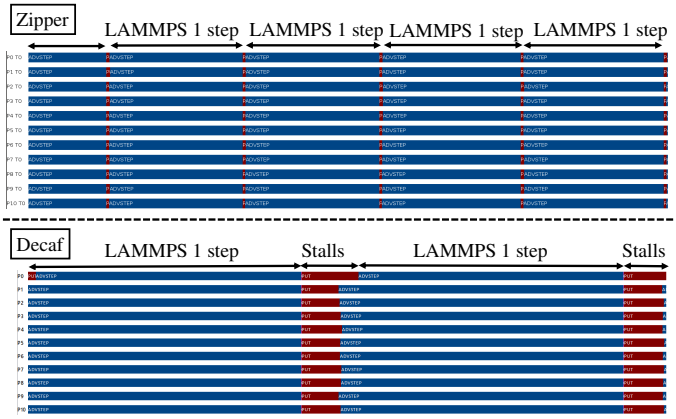


Figure 19: Trace comparison between Zipper and Decaf for the LAMMPS application on 13,056 cores. This figure shows a snapshot of 9.1 seconds when using 13,056 cores, which is taken from the experiment shown in Figure 18.

6.3.2 The LAMMPS application. Figure 18 shows the experimental results for the LAMMPS workflow application. Again, we perform weak scaling experiments. Figure 18 shows that Flexpath scales well from 204 to 3,264 cores but is 7.1 times slower than Zipper. Because the data size in LAMMPS does not reach the integer limit, we are able to execute Decaf on 6,528 and 13,056 cores successfully without integer overflows. From the figure, we can see that Decaf scales greatly from 204 to 1,632 cores, but becomes 128% slower from 1,632 to 6,528 cores. Eventually, its end-to-end time increases by 177% from 6,528 to 13,056 cores.

To study why Decaf is 2.2 times slower than Zipper in the largest experiment, we specifically collect two very large traces for Decaf and Zipper using 13,056 cores, respectively. Visualizing the large-scale trace itself requires us to use a dedicated compute node from the *Stampede2* HPC system for 2 hours.

Figure 19 shows a snapshot of the two traces in an interval of 9.1 seconds. During the same time interval, LAMMPS using Zipper runs around 4.4 time steps. On the other hand, LAMMPS using Decaf runs around 2 time steps. Notice that the Decaf trace has a significant stall time at the end of each step. Also, the LAMMPS simulation time using Decaf becomes much longer than that using Zipper. In this LAMMPS workflow experiment, each LAMMPS process generates approximately 20MB of data in each time step. While Decaf directly sends a message of 20MB to destination processes, Zipper divides the contiguous 20MB data into many small blocks of size 1.2MB. Such an asynchronous fine-grain-block data transfer method has managed to keep network traffic more balanced with lesser interference to the LAMMPS simulation processes.

7 CONCLUSION

This work studies the important class of scientific workflows that combine large-scale simulations with big data analysis by carrying out performance analysis and optimization on the present I/O and data transfer libraries. Our trace analyses reveal that there are significant performance inefficiencies in the current practice (such as remote server and metaserver read/write time, coarse-grain critical sections, interlock between applications, barriers, and application stalls). With the aim of minimizing the end-to-end time of scientific workflows, we propose to combine the abstraction of pipelining parallelism with the abstraction of fine-grain task parallelism to totally intertwine the simulation and analysis applications such that the time-to-solution is merely one stage of time. A new Zipper runtime system has been designed and implemented. Supported by both an analytical performance model and empirical experiments, we show that the Zipper system can obtain the fastest end-to-end time, which almost reaches the lower bound of the simulation-only time. In addition, the concurrent data transfer optimization can reduce the stall time of the simulation application when the simulation is coupled with a relatively slow data analysis. Our experiments with the real-world CFD and LAMMPS workflows show that the Zipper approach is able to outperform the Decaf method — which is the fastest one among seven modern methods — by up to 2.2 times. A set of subsequent traces also reveal that the reduced idle/stall time, the lesser interference with the simulation time, and the full overlapping of all workflow stages have contributed the most to Zipper's enhanced end-to-end workflow time.

ACKNOWLEDGMENT

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by the NSF Grant# ACI-1548562. It is also supported by the NSF Grant# 1513201. We thank Dr. Luoding Zhu for providing the parallel CFD application. We also thank the DataSpaces team and the Flexpath team for their support of using their software. In particular, we would like to thank our shepherd, Dr. Gerald Lofstead, for providing many suggestions and guiding the revision of our paper.

REFERENCES

[1] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. 2009. Extending I/O through high performance data services. In *IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*. IEEE, 1–10.

[2] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. 2010. Datastager: Scalable data staging services for petascale applications. *Cluster Computing* 13, 3 (2010), 277–290.

[3] Fatma Alali, Fabrice Mizero, Malathi Veeraraghavan, and John M Dennis. 2017. A measurement study of congestion in an InfiniBand network. In *Network Traffic*

Measurement and Analysis Conference (TMA), 2017. IEEE, 1–9.

[4] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, et al. 2016. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 577–597.

[5] J.C. Bennett, H. Abbasi, P.T. Bremer, R. Grout, A. Gyulassy, T. Jin, et al. 2012. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 1–9.

[6] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path architecture: Enabling scalable, high performance fabrics. In *The 23rd IEEE Annual Symposium on High-Performance Interconnects (HOTI)*. IEEE, 1–9.

[7] J Chen, A Choudhary, S Feldman, B Hendrickson, CR Johnson, R Mount, V Sarkar, V White, and D Williams. 2013. Synergistic challenges in data-intensive science and exascale computing. *DOE ASCAC Data Subcommittee Report, Department of Energy Office of Science* (2013).

[8] DataSpaces Project. 2018. http://dataspaces.org. (2018).

[9] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. 2014. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 246–255.

[10] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.

[11] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. DataSpaces: An interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (2012), 163–181.

[12] S Dormido-Canto, J Vega, JM Ramírez, A Murari, R Moreno, JM López, A Pereira, and JET-EFDA Contributors. 2013. Development of an efficient real-time disruption predictor from scratch on JET and implications for ITER. *Nuclear Fusion* 53, 11 (2013), 113001.

[13] Matthieu Dreher and Tom Peterka. 2017. *Decaf: Decoupled dataflows for in situ high-performance workflows*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[14] Greg Eisenhauer, Matthew Wolf, Hasan Abbasi, and Karsten Schwan. 2009. Event-based systems: opportunities and challenges at exascale. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM.

[15] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen. 2011. The Paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 89–96.

[16] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. 2016. Big data, simulations and HPC convergence. In *Workshop on Big Data Benchmarks*. Springer, 3–17.

[17] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. 2007. Examining the Challenges of Scientific Workflows. *Computer* 40, 12 (Dec 2007), 24–32.

[18] E. Gonsiorowski, C. D. Carothers, J. LaPre, P. Heidelberger, C. Minkenberg, and G. Rodriguez. 2017. Using quality of service lanes to control the impact of RAID traffic within a burst buffer. In *2017 Winter Simulation Conference (WSC)*. 932–943. https://doi.org/10.1109/WSC.2017.8247844

[19] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced features of the message-passing interface*. MIT press.

[20] Zhaoli Guo and Chang Shu. 2013. *Lattice Boltzmann method and its applications in engineering*. World Scientific.

[21] Huateng Huang and L Lacey Knowles. 2014. Unforeseen consequences of excluding missing data from next-generation sequences: Simulation study of RAD sequences. *Systematic biology* (2014), 1–9.

[22] Intel. 2015. *Intel Omni-Path Fabric Suite Fabric Manager GUI User Guide*.

[23] Intel. 2018. Intel Trace Analyzer and Collector. (2018). https://software.intel.com/en-us/intel-trace-analyzer

[24] Selim Kalayci, Gargi Dasgupta, Liana Fong, Onyeka Ezenwoye, and Seyed Masoud Sadjadi. 2010. Distributed and Adaptive Execution of Condor DAGMan Workflows.. In *SEKE*. 587–590.

[25] Bongjae Kim and Jeong-Dong Kim. 2017. Dynamic QoS Scheme for InfiniBand-Based Clusters. In *Advances in Computer Science and Ubiquitous Computing*, James J. (Jong Hyuk) Park, Yi Pan, Gangman Yi, and Vincenzo Loia (Eds.). Springer Singapore, Singapore, 573–578.

[26] Feng Li and Fengguang Song. 2017. A Real-Time Machine Learning and Visualization Framework for Scientific Workflows. In *Practice & Experience in Advanced Research Computing (PEARC-2017)*. ACM, New Orleans, LA.

[27] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, and Ron Oldfield. 2014. Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[28] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience* 18, 10 (2006), 1039–1065.

[29] John L Lumley. 2007. *Stochastic tools in turbulence*. Courier Corporation.

[30] Ewing L Lusk, Steve C Pieper, Ralph M Butler, et al. 2010. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review* 17, 1 (2010), 30–37.

[31] Kwan-Liu Ma. 2009. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE* 29, 6 (2009), 14–19.

[32] Shahrbanou Madadgar, Hamid Moradkhani, and David Garen. 2014. Towards improved post-processing of hydrologic forecast ensembles. *Hydrological Processes* 28, 1 (2014), 104–122.

[33] Takemasa Miyoshi, Masaru Kunii, Juan Ruiz, Guo-Yuan Lien, Shinsuke Satoh, Tomoo Ushio, Kotaro Bessho, Hiromu Seko, Hirofumi Tomita, and Yutaka Ishikawa. 2016. "Big Data Assimilation" revolutionizing severe weather prediction. *Bulletin of the American Meteorological Society* 97, 8 (2016), 1347–1354.

[34] P. Nagar, F. Song, L. Zhu, and L. Lin. 2015. LBM-IB: A Parallel Library to Solve 3D Fluid-Structure Interaction Problems on Manycore Systems. In *Proceedings of the 2015 International Conference on Parallel Processing (ICPP'15)*. IEEE.

[35] National Academies of Sciences, Engineering, and Medicine. 2016. *Future Directions for NSF Advanced Computing Infrastructure to Support U.S. Science and Engineering in 2017-2020*. Washington, DC: The National Academies Press. https://doi.org/10.17226/21886

[36] PAPI project. 2018. http://icl.utk.edu/papi/. (2018).

[37] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Commun. ACM* 58, 7 (2015), 56–68.

[38] S-A Reinemo, Tor Skeie, Thomas Sodring, Olav Lysne, and O Trudbakken. 2006. An overview of QoS capabilities in InfiniBand, advanced switching interconnect, and Ethernet. *IEEE Communications Magazine* 44, 7 (2006), 32–38.

[39] Denis Ricot, Virginie Maillard, and Christophe Bailly. 2002. Numerical simulation of unsteady cavity flow using Lattice Boltzmann Method. In *8th AIAA/CEAS Aeroacoustics Conference & Exhibit*. 2532.

[40] Joerg Schumacher. 2001. Derivative moments in stationary homogeneous shear turbulence. *Journal of Fluid Mechanics* 441 (2001), 109–118.

[41] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[42] Angela B Shiflet and George W Shiflet. 2014. *Introduction to computational science: Modeling and simulation for the sciences*. Princeton University Press.

[43] Q. Sun, M. Romanus, T. Jin, H. Yu, P. Bremer, S. Petruzza, S. Klasky, and M. Parashar. 2016. In-staging data placement for asynchronous coupling of task-based scientific workflows. In *International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2–9.

[44] Alexander S Szalay. 2013. From Large Simulations to Interactive Numerical Laboratories. *IEEE Data Eng. Bull.* 36, 4 (2013), 41–53.

[45] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*. ACM, 23–32.

[46] V. Vishwanath, M. Hereld, M.E. Papka, R. Hudson, G.C. Jordan IV, and C Daley. 2011. In Situ Data Analysis and I/O Acceleration of FLASH Astrophysics Simulation on Leadership-Class System Using GLEAN. In *Proc. SciDAC, Journal of Physics: Conference Series*.

[47] VisIt. 2018. https://visit.llnl.gov. (2018).

[48] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, Paul Fisher, et al. 2013. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic acids research* 41, W1 (2013), W557–W561.

[49] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. 2013. Swift/T: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 95–102.

[50] Fan Zhang. 2015. *Programming and runtime support for enabling data-intensive coupled scientific simulation workflows (Phd dissertation)*. Rutgers The State University of New Jersey-New Brunswick.

[51] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. 2010. PreDatA-preparatory data analytics on peta-scale machines. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[52] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.A. Nguyen, J. Cao, H. Abbasi, and S. Klasky. 2013. FlexIO: I/O middleware for location-flexible scientific data analytics. In *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 320–331.

[53] Luoding Zhu, Derek Tretheway, Linda Petzold, and Carl Meinhart. 2005. Simulation of fluid slip at 3D hydrophobic microchannel walls by the lattice Boltzmann method. *J. Comput. Phys.* 202, 1 (2005), 181–195.